Implementation of the Kauffman Polynomial in SageMath

Antonio De Lucreziis

Abstract. In this project we implement the Kauffman polynomial in SageMath (Python).

Contents

1. Introduction 1
  1.1. The Kauffman Polynomial 1
  1.2. Computational Knot Theory 2
    1.2.1. PD codes 2
    1.2.2. Signed Gauss Codes 4
    1.2.3. Link reconstruction from code 6
      1.2.3.1. Linear Integer Programming 6
      1.2.3.2. Planar Graph Embeddings 6
  1.3. Algorithm for computing the Kauffman Polynomial 7
  1.4. Python Implementation 8
    1.4.1. Signed Gauss Codes 9
      1.4.1.1. Writhe 9
      1.4.1.2. Standard Unknot 10
      1.4.1.3. Crossing Splices 10
2. Appendix 12

1. Introduction

Actually we don't like Python so we will be using Rust and then write bindings for Python that can be used in SageMath.

1.1. The Kauffman Polynomial

The Kauffman polynomial L is a two-variable polynomial invariant of unoriented knots and links in 3-dimensional space. It is defined using *Skein relations*, more precisely an implicit functional equation.

The defining axioms of the Kauffman polynomial are the following, given a link diagram K we have $L_{K(a,z)} \in \mathbb{Z}[a, a^{-1}, z, z^{-1}]$ and:

i) If K and K' are two equivalent up to regular isotopy, then $L_K(a, z) = L_{K'}(a, z)$.

ii) We have the following identities:
   - $L\big(\diagup\!\!\!\!\diagdown\big) + L\big(\diagdown\!\!\!\!\diagup\big) = z\Big(L\big(\asymp\big) + L\big(\,)\;(\,\big)\Big)$
   - $L\big(\bigcirc\big) = 1$

- $L\Big(\quad\Big) = aL\Big(\quad\Big)$
- $L\Big(\quad\Big) = a^{-1}L\Big(\quad\Big)$

We will later be seeing that the Kauffman polynomial can be defined in a more explicit way, using a recursive definition that is the one we will be using to derive our algorithm.

1.2. Computational Knot Theory

The first problem in computational knot theory is to find a good representation for knots and links. There are various common representations, such as:

- **Gauss codes**:

  This is very simple to generate, we just need to label each crossing with a number and then write down the sequence of numbers in the order they appear by walking along the knot. This has the problem that it is not unique, for example it does not distinguish between the trefoil knot and its mirror image.

- **Signed Gauss codes**:

  This is an improvement over the previous representation, on the second occurrence of a number we use a + or − sign to indicate the handedness of the crossing.

- **Planar diagram codes (PD codes)**, this is the one we will be using in this project:

  The PD code for a link is generated by labelling each arc of the link with a number. Then we choose a starting point and walk along the link, when we pass at an over-crossing for the current strand we write down a 4-uple of counter-clockwise numbers for the arc incident to the crossing.

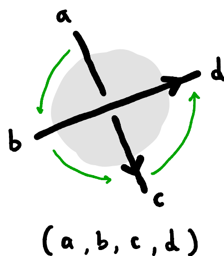
There are also other codes like **Braid representations** and **DT (Dowker-Thistlethwaite) Codes** we will not be using in this project.

1.2.1. PD codes

Reference: PD notation article from KnotInfo

The PD code for a link is generated by labelling each arc of the link with a number. Then we choose a starting point for each component and process each component in order.

For each component we walk along it from the starting point in the component direction.



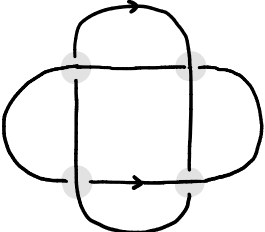
When we pass at a crossing that is an over-crossing for the current strand we write down a 4-uple of counter-clockwise numbers for the arc incident to the crossing starting from the *entering under-crossing* arc.

Algorithm:

```
Input: An oriented link diagram with starting points on each component
Output: List<(Nat, Nat, Nat, Nat)>

- Choose an ordering for the components and starting point for each component
- Label each arc with a number
- For each component:
  - Walk along it from the starting point in its orientation
  - At each crossing, when at an over-crossing for the current strand
    - Write down a 4-uple of counter-clockwise numbers for the arc incident
      to the crossing starting from the entering under-crossing arc
```

Let's see an example of how to construct the PD code for the following link diagram



First let's choose a starting point for each component and label accordingly the arcs of the link. We will use the following convention:

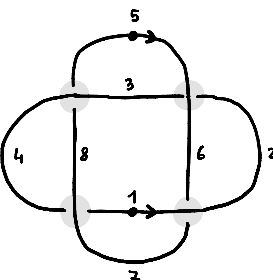
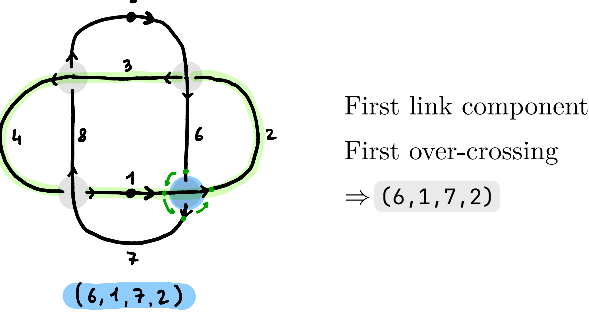
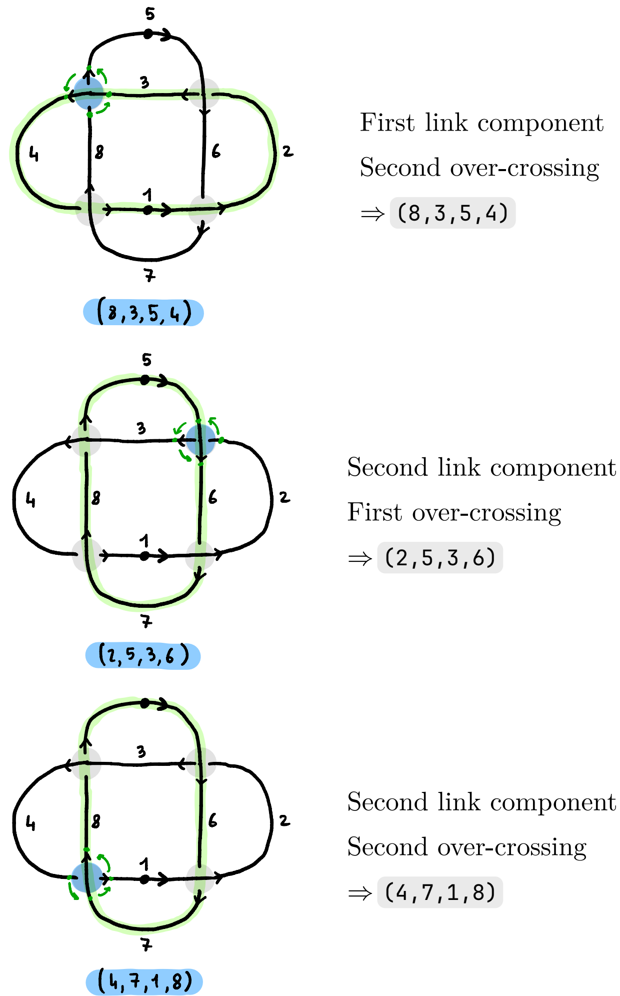


FIGURE 3. Oriented link with starting points and edge labels.

Now we can start processing each component of the link by walking along it in its orientation and writing down the over-crossings we encounter.



First link component
First over-crossing
⇒ `(6,1,7,2)`



Every directed crossing appears only once as an over-crossing so this algorithm terminates when all crossings have been visited. So the final PD code for this link is

```
[(6,1,7,2), (8,3,5,4), (2,5,3,6), (4,7,1,8)]
```

1.2.2. Signed Gauss Codes

Gauss originally developed a notation called **Gauss codes** based on labelling each crossing of a knot with a number and keeping track of when we walk an over-crossing or an under-crossing using a sign. This produces a list of numbers where each number appears exactly twice with different signs. This has a few problems like the fact that this doesn't distinguish a knot vs its mirror.

This is solved by **Signed Gauss Codes**[1] where we also store the information about the handedness of a crossing.

More precisely this is constructed by the following: for each component we walk along it, when we pass at a crossing we write a tuple $(\pm i, \pm 1)$ where the first component is the index of the crossing with a sign indicating if this is an over-crossing or under-crossing, the second sign (that can be added in a second pass over the loop) is given by the handedness of the crossing using the following convention

[1] Gauss Notation on Wikipedia

$$\varepsilon(\times) = +1 \qquad \varepsilon(\times) = -1$$

Algorithm:

```
Input: An oriented link diagram with starting points on each component
Output: List<List<(Int, Int)>>

- Label each crossing with a number in order
- For each component:
  - Walk along it from the starting point in its orientation
  - At each crossing, write a tuple with components
    - $+i$ or $-i$ if this is an over-crossing or under-crossing
    - $+1$ or $-1$ if this is a left-handed or right-handed
```

Converting one code to the other is not too much work as one just need to first do a labelling step to convert crossing labels and then convert the over/under-strand and left/right-handedness relations between the two notations.

For example the *Knot Theory code for the Kauffman polynomial* converts the **pd notation** to **signed gauss notation** as this is better suited for doing manipulations directly on the crossings.

- Crossing switch is just two sign swaps on each of the two occurrences of the over and under strand
- Splicing is just a matter of splitting and rejoining lists correctly, for example let's see what happens in the case of an *horizontal splice*. There are two cases based on the orientation of the strands:
  - If the splice happens on a **self-crossing** (crossing with a part of the same curve) then we apply the following modification to the link, the rest remains the same except for the curve containing the spliced crossing that is changed as follows

    $$\left[\dots \ell_1^+ \dots \times \dots \ell_2^+ \dots \times \dots \ell_3^+ \dots\right] \mapsto \left[\dots \ell_1^+ \dots \asymp \dots \ell_3^+ \dots\right], \left[\asymp \dots \ell_2^+ \dots\right]$$

    $$\left[\dots \ell_1^+ \dots \times \dots \ell_2^+ \dots \times \dots \ell_3^+ \dots\right] \mapsto \left[\dots \ell_1^+ \dots \asymp \dots \ell_2^- \dots \asymp \dots \ell_3^+ \dots\right],$$

    here by "$\dots \ell_i^+ \dots$" we mean a part of the crossing list and "$\dots \ell_i^- \dots$" is the same list reversed.
  - Otherwise if the splice happens on a crossing **between strands of different curves**

    $$\left[\dots\left[\dots \ell_1^+ \dots \times \dots \ell_2^+ \dots\right] \dots \left[\dots \ell_3^+ \dots \times \dots \ell_4^+ \dots\right] \dots\right]$$
    $$\downarrow$$
    $$\left[\dots\left[\dots \ell_1^+ \dots \asymp \dots \ell_4^+ \dots \ell_3^+ \dots \asymp \dots \ell_2^+ \dots\right] \dots\right]$$

    or in case of the other orientation

$$\Big[\dots\Big[\dots\ell_1^+\dots \times \dots\ell_2^+\dots\Big]\dots\Big[\dots\ell_3^+\dots \times \dots\ell_4^+\dots\Big]\dots\Big]$$

$$\downarrow$$

$$\Big[\dots\Big[\dots\ell_1^+\dots \asymp \dots\ell_3^-\dots\ell_4^-\dots \asymp \dots\ell_2^+\dots\Big]\dots\Big]$$

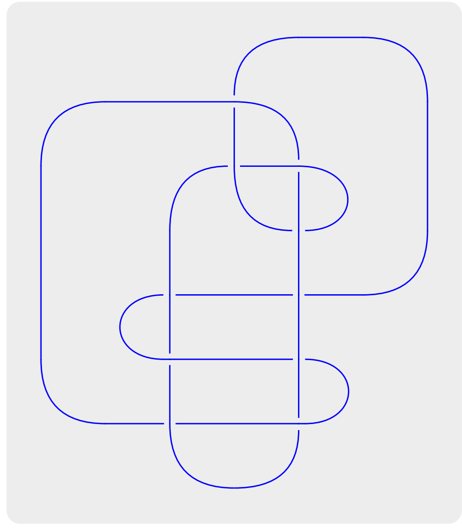
1.2.3. Link reconstruction from code

We briefly mention that reconstructing a link from a PD code is not trivial and there are various approaches that can be used for this task.

◇ Linear Integer Programming

For example the KnotTheory package in Sage has a `Link.plot()` method that thats a link that can be constructed using a PD code and then plots it as follows

```
# The "monster" unknot
L = Link([[ 3,  1,  2,  4], [ 8,  9,  1,  7], [ 5,  6,  7,  3], [ 4, 18,  6,  5],
          [17, 19,  8, 18], [ 9, 10, 11, 14], [10, 12, 13, 11], [12, 19, 15, 13],
          [20, 16, 14, 15], [16, 20, 17,  2]])
L.plot()
```



Sage internally uses a *mixed integer linear programming* (MILP) solver to generate a knot diagram from a PD code. Another library called Spherogram instead uses *network flows*. The problem here is to find an orthogonal presentation for the link with the *minimum number of left and right bends*[2].

◇ Planar Graph Embeddings

Another approach used by KnotFolio is based on Tutte embeddings. A **Tutte embedding or barycentric embedding** of a simple, 3-vertex-connected, planar graph is a crossing-free straight-line embedding with the properties that the outer face is a convex polygon and that each interior vertex is at the average (or barycenter) of its neighbors' positions.

[2] A better heuristic for area-compaction of orthogonal representations

This condition that every point is the average of its neighbors can be easily expressed as a system of linear equations where some points on a chosen outer face have been fixed. When the graph is planar and 3-vertex-connected the linear system is non degenerate and has a unique solution.

1.3. Algorithm for computing the Kauffman Polynomial

Let's now recap the main formal algorithm for computing the Kauffman polynomial.

Definition 1. Let K be an un-oriented link. We denote by $\hat{K}(p)$ the **standard unknot** for K where p is a *directed starting point*. This is built by considering the planar shadow U of K and walking along U starting from p and making each crossing an over-crossing when passing on it the first time.

TODO: Disegnini

Definition 2. Let's give a name to the following knot modifications for K near a specific crossing i

K	S_iK	E_iK	e_iK
original	*switch*	*h-splice*	*v-splice*

Let now K be an oriented link with n components so $K = K_1 \cup ... \cup K_n$.

Definition 3. Let K abd $\lambda = (\lambda_n, ..., \lambda_0)$ a sequence of indices of crossing of K and let i be an index of one of the crossings, let's define the following actions

- $A_i^\lambda K \coloneqq E_i S_{\lambda_i} \cdots S_{\lambda_0} K$
- $B_i^\lambda K \coloneqq e_i S_{\lambda_i} \cdots S_{\lambda_0} K$
- Then let λ be a sequence of indices that bring K to $\hat{K}$ so that $\hat{K}(\lambda) \coloneqq S_{\lambda_n} \cdots S_{\lambda_0} K$ and define

$$\sum_K(\lambda) = \sum_{i=0}^{n} (-1)^i \big(L\big(A_i^\lambda K\big) + L\big(B_i^\lambda K\big)\big)$$

$$\Omega_K(\lambda) = (-1)^{|\lambda|+1} L_{\widehat{K}(\lambda)} + z \sum_K(\lambda)$$

Algorithm. Here follows the algorithm that computes $L_K(a, z)$.

i) If $K = \hat{K}$ is a *standard unknot* then $L_K(a, z) \coloneqq a^{w(K)}$

ii) If K_1 *overlies* K_2, let $d \coloneqq (a + a^{-1})/z - 1$ and then

$$L(K_1 \cup K_2) \coloneqq dL(K_1)L(K_2)$$

iii) If $K = K_1 \cup \cdots \cup K_n$

- If a K_i *overlies* another another component than apply (ii).
- If no K_i *overlies* all others let $p_1, ..., p_n$ be *directed starting points* on $K_1, ..., K_n$ and let $\overline{p}_i$ be the same *directed starting point* with the opposite direction of p_i on K_i. Let $\lambda(p_i)$ the sequence of under-crossings of K_i with $K - K_i$ so that $\hat{K}(\lambda(p_i)) =$

$K_i \sqcup (K - K_i)$ so that K_i *overlies* the rest of the components. At this point we can define L_K as

$$L_K(a,z) := \frac{1}{2n}\left[\sum_{i=1}^{n}\sum_{q=p_i,\overline{p}_i}\left((-1)^{|\lambda(q)|+1} d L_{K_i} L_{K-K_i} + z\sum_{K}(\lambda(q))\right)\right]$$

- If K is a single component then let p be a directed starting point on K and $\overline{p}$ the one with opposite direction. Let $\lambda(p)$ the switching sequence that brings it to the standard unknot $\hat{K}$ and define

$$L_K(a,z) := \frac{1}{2}\left[\sum_{q=p,\overline{p}}\left((-1)^{|\lambda(q)|+1} L\Big(\hat{K}(\lambda(q))\Big) + z\sum_{K}(\lambda(q))\right)\right]$$

1.4. Python Implementation

The approach has been a mix of bottom-up and top-down. First we defined a couple of classed `SignedGaussCode` and `PDCode` to work with these codes and easily convert between each other.

This initial implementation uses `SignedGaussCodes` as they are easier to work with when working with crossing switches and splices but with some modifications the code could be adapted to work directly on `PDCode` provided of some efficient implementations of `splice_h` and `splice_v` methods.

1.4.1. Signed Gauss Codes

We are now going to walk thorough the class that lets use work nicely with **Signed Gauss Codes**. The the classes we are going to use are all *frozen data-classes* to ensure immutability and enforce a more functional programming style.

```
Sign = typing.Literal[+1, -1]

@dataclass(frozen=True)
class SignedGaussCodeCrossing:
    id: int
    over_under: Sign
    handedness: Sign

    def is_over(self) → bool: ...
    def is_under(self) → bool: ...
    def is_left(self) → bool: ...
    def is_right(self) → bool: ...
    def opposite(self) → SignedGaussCodeCrossing: ...

    def __repr__(self): ...
```

```
@dataclass(frozen=True)
class SignedGaussCode:
    components: list[list[SignedGaussCodeCrossing]]

    def writhe(self): ...
    def reverse(self): ...
    def mirror(self): ...
    def to_std_unknot(self) → SignedGaussCode: ...
    def std_unknot_switching_sequence(self) → list[int]: ...
    def apply_switching_sequence(self, seq: list[int]) → SignedGaussCode: ...
    def splice_h(self, id: int): ...
    def splice_v(self, id: int): ...
    def switch_crossing(self, id: int): ...

    def __repr__(self): ...
```

◇ **Writhe**

One of the first important things we need is to compute the **writhe** $w(K)$ of a link, this can easily be done with signed gauss codes as its a list of tuples where the second entry is the crossing sign. Let L be an oriented link with components $C_1, ..., C_k$ each with crossings $c_{i,j}$ with $i = 1, ..., k$ and $j = 1, ..., |C_i|$.

Let's notice that here each crossing appears twice, once as over-crossing and once as an under-crossing this is the reason for the $1/2$ in the following formula. By $\varepsilon(c)$ we refer to the sign (or handedness) of the crossing at c.

$$w(L) = \frac{1}{2} \sum_{c \text{ crossing}} \varepsilon(c) \quad \rightsquigarrow$$

```
def writhe(self):
    return sum(
        c.handedness # ⇒ +1 or -1
        for component in self.components
        for c in component
    ) // 2
```

◇ Standard Unknot

The next building block for computing the Kauffman polynomial is detecting and computing the **standard unknot or unlink**. Formally this is done by taking the *planar shadow* and a directed starting point on it. Then we can walk along the shadow and make each crossing an over-crossing when passing on it on the first time.

On the other hand our algorithm directly works with switching sequences λ that bring a knot K to its standard unknot $\hat{K}$. We wrote methods to directly compute and apply these switching sequences.

```python
def std_unknot_switching_sequence(self) -> list[int]:
    visited_crossings: set[int] = set()
    switched_crossings: list[int] = []

    for component in self.components:
        for crossing in component:
            if crossing.id not in visited_crossings:
                if crossing.is_under():
                    switched_crossings.append(crossing.id)
                visited_crossings.add(crossing.id)

    return switched_crossings
```

The `std_unknot_switching_sequence` method just walks along each component in its orientation marking what switches have to be made to bring that link to its standard unknot.

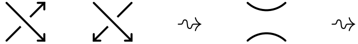
```python
def apply_switching_sequence(self, seq: list[int]) -> SignedGaussCode:
    return SignedGaussCode(
      [
        [
            crossing.opposite() if crossing.id in switching_sequence else crossing
            for crossing in component
        ]
        for component in self.components
      ]
    )
```

Applying a switching sequence is just a matter of walking along the crossings and flipping the crossings that are in the sequence. This is also how the `switch_crossing(id: int)` method works.

◇ Crossing Splices

The splicing code is more involved due to the number of cases to analyze, let's first see formally what we need to do.

We have all the following cases, first we can assume the *entering over strand* is in the top left corner of a diagram (this can be done by applying locally a small isotopy). So we have 2 cases for the crossing sign



So the final code is just a conversion of all this cases to list slicing and re-joining with the appropriate crossings removed.

```
def splice_h(self, id: int):
    raise NotImplementedError("Splicing not implemented yet")

def splice_v(self, id: int):
    raise NotImplementedError("Splicing not implemented yet")
```

2. Appendix

All combinations of `skein-generic` typst function